UNIVERSITY OF STAVANGER

Dependability Evaluation of the Spread/Distributed Autonomous Replication Management Framework in a Realistic Deployment Scenario

by

Kristian Nesvik Andersen

A thesis submitted in fulfillment for the
degree of master

in
computer science
faculty of science and technology

June 2010

"The time of getting fame for your name on its own is over. Artwork that is only about wanting to be famous will never make you famous. Any fame is a by-product of making something that means something. You don't go to a restaurant and order a meal because you want to have a shit."

-Banksy

Abstract

DARM is a middleware framework for managing and distributing services through replication. It is based on the underlying group communication framework Spread, which delivers fast and reliable group communication, with membership management, where DARM use multiple separate groups to separate different areas of functionality. DARM is designed to tolerate node faults and network partitioning, while ensuring availability and reliability to multiple services on each network segment through recovery procedures.

In this thesis is presented an approach to determine dependability attributes of a fully deployed DARM system, with multiple realistic services included. Also presented is the work that has been done to prepare and extend the DARM framework with toolskits that are considered indispensable to distributed services. DARM is compared with similar solutions, and presented as a potential framework for acting as a cloud computing middleware, supporting both cloud infrastructure and service development.

Results of this thesis indicate that DARM is reliable, performing consistently with high grades of availability and MTBF. Remaining deficiencies are reduced to acceptable levels.

Acknowledgements

The work with *DARM* has been very interesting and challenging. I will be sure to follow this project in the future.

First I want to thank my supervisor Hein Meling (associate professor at UiS) for introducing me to DARM and the challenges there. He has been available for support throughout the entire working period of this thesis. Joakim L. Gilje has proven invaluable with all his knowledge about core DARM behaviour. Joakim has also provided direct help with adjusting the source code of DARM. I also want to thank Theodor Ivesdal, the administrator of the Linux-lab at UiS. Besides making sure the environment being up and running, he has presented useful tools and procedures for the lab.

Contents

List of Figures

List of Tables

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
CEAS	**C**ross **E**ntropy **A**nt **S**ystem
CPU	**C**entral **P**rocessing **U**nit
DARM	**D**istributed **A**utonomous **R**eplication **M**anagement
DHCP	**D**ynamic **H**ost **C**onfiguration **P**rotocol
GCS	**G**roup **C**ommunication **S**ystem
GMI	**G**roup **M**ethod **I**nvocation
IO	**I**nput **O**utput
JGCS	**J**ava **G**roup **C**ommunication **S**ystem
JNI	**J**ava **N**ative **I**nterface
MTBF	**M**ean **T**ime **B**etween **F**ailure
NTP	**N**etwork **T**ime **P**rotocol
QoS	**Q**uality **o**f **S**ervice
RMI	**R**emote **M**ethod **I**nvocation
SSH	**S**ecure **SH**ell
UiS	**U**niversitetet **i** **S**tavanger

Chapter 1

Introduction

Distributed Autonomous Replication Management (DARM) is a system for decentralized management of replicated distributed services. It improves dependability characteristics by focusing on deployment and operational aspects, where the gain in terms of improved dependability is likely to be the greatest. By distributing its services in replica, over geographically separated locations, DARM ensures that any service offered are highly available, reliable, and independent of concurrent replica failures. Network failures which causes partitioning are also countered as DARM ensures that any network partition is maintained with service requirements. DARM completely automates the replication task, handling placement of replica without human interference. The resilience level of involved services are ensured on each partition. Each partition elects a group leader who takes responsibility for any fault treatment. In case of a disrupted leader, a new leader is chosen. Also the merging of two or more network partitions, back to one partition, is fully automated where any excessive replica in the new partition are removed. By distributing recovery decisions to each individual network partition, the need for a centralized manager with global information about all groups is eliminated. The complexity of the infrastructure and the communication overhead is reduced. One of the main goals of DARM is to minimize the time spent in a state of reduced fault resilience; the time between an occurrence of one or more faults till being recovered to the same given resilience level. Unlike most of the other replication management systems, DARM accomplish its goals thanks to a underlying group communication scheme (GCS) called Spread. Spread delivers lightweight, fast, and reliable group communications, enabling DARM to make its decisions while consuming a low degree of networking resources. Since DARM builds on Spread, the same system and failure model as the one assumed by Spread, is also assumed for DARM. Spread allows processors of a network to connect to each other in a group. When inside a group, group broadcast messages are automatically received, and messages may be sent one by one, or broadcast, to other

members of the same group. DARM use Spread to form multiple groups. One group is used by each instance of DARM to communicate information about the current network state. Also a group is formed for each service running on the DARM network. The later groups are used by the services themselves to communicate service specific messages. In GCS context a group is denoted as a *view*. This view is concurrently identical to every member of the group, and functions as an overview to the current status of the group. The view holds a reference to all its members. In case of members joining or leaving a group, *viewchanges* are triggered by the GCS to ensure that the view is consistent for every member. We say that the member installs a new view whenever a viewchange occurs.

DARM can be seen as a system, or *middleware*, used for building *cloud services*. Cloud based development was originally lacking proper development support. Development for both infrastructure and distributed services is now supported. Mainly two attributes of DARM makes it stand out in the cloud computing scene; its decentralized architecture with responsibility distributed to each active partition of the network, and its appliance of GCS as a foundation. Traditionally the centralized approach has been used to develop cloud infrastructure. A centralized architecture implies that a parent processor or a system administrator, the central manager, dedicates to ensuring availability and reliability. This approach introduces not only a bottleneck to the system, as the central manager may experience overloads, but also a single point of failure, where a fault on the central manager would inflict a paralysis to the system's ability to counter any faults.

Since not much measurement and development have been done on DARM after its first operational version was developed in 2007, the work done here have aimed to expand these fields. In order to achieve easier service development, a platform promoting and easing such work was called for. Such platform did not originally exist on DARM, and it should not only promote developing new service applications, but it was also desired to open for direct porting of existing service applications from GCSs similar to Spread. We shall look into a generic group communication interface that has already earned a reputation among such systems, along with useful tools for distributed services.

The contribution to the DARM framework that has been done through the work that is presented here, constitutes in the following areas:

- DARM did not offer a very supportive **platform** for developing distributed services. This was solved through implementing Java Group Communication System (JGCS); a set of interfaces that generalize the terminology and approaches of methods available to distributed services on the Spread/DARM platform. This helps a developer as the standard procedures of GCS based service development

are made available. A very positive side effect of this work, is that also direct porting of existing distributed services based on JGCS is possible; requiring a minimum of changes to the original source code. Java based development is now well supported in DARM. Multiple approaches to developing distributed systems have been made available.

- DARM did also lack a predefined way to achieve **load balancing**. In distributed services load balancing is typically achieved through using *anycast* messaging. Anycast is when a message is sent, from a client, to one random member of the service group instead of the corresponding procedure of *multicast* which broadcast the message to every member of the group. A predefined API called Group Method Invocation (GMI) enables, among other techniques, anycast. This is solved through the use of a proxy. The proxy unites the services and enables clients to connect and interact with services as one unit, rather than having to treat the service as a group of service instances. GMI has been implemented in the Spread/DARM framework.

- Only one example of **realistic services** on DARM was known about when this work started. More such realistic services was needed. JGCS and GMI was used to develop and port multiple new and existing distributed services in DARM. These include an allocator service, a mail service, and a somewhat simpler event logging service. Direct porting was used on the allocator service, while the mail service and the event logger was built more or less from bottom up. The services have been designed to be able to log all its relevant events to separate files.

- **Tools and procedures for experimenting** with DARM where not defined. Techniques for such work needed to be properly tested before any real experiments could proceed. Bash scripts proved useful for this work and where examined and experimented with before any organized experiments could take place. Bash scripts allow to store and run sequences of commands, and to use the typical logical operators of a programming language. Sometimes it performs a little slow, but it generally meets the accuracy requirements set for experimenting done here.

- DARM was still prone to errors, leaving much work to be done in **identifying and correcting erroneous behaviour**. The most serious errors hindered any experiments to produce results of significant value. Many unwanted issues in the DARM framework was identified and corrected. Most of this work will not be presented in this text, yet it is emphasized that without it any experimenting would prove faulty and useless.

- No **full scale experiments** had been run on the DARM framework in advance to this work. Only simpler experiments with individual fault injections or network

partitions have been presented. The experiments where aimed at 2 important tasks; functioning as a quality insurance, and combing out remaining deficiencies in DARM. Deficiencies have been reduced to a level where any remaining such are only observed in very rare occasions. Typically a multi plum of 1000 correct deployments and recoveries are allowed between observations.

- During experiments, each service is **logging** its events. Log parsers have been developed to collect and analyse these log files post experiment. The goal of the logging has been to produce a single log file, out of larger collections of log files produced by each replica, that are chronological sorted to resemble a single time line of events for the given experiment. The time line is again used by analysing tools to decide accurate statistics of the experiment. The log parsers have been developed in Java, and they are indispensable in the work of analysing larger experiments. The performance of the log parsers, processing several MBs of log files in a matter of seconds, have proven exceptional.

- Techniques for defining **dependability characteristics** of fault resilience replicated systems where defined for frameworks similar to DARM. Full scale experiments where configured and used to produce the data required to calculate the dependability characteristics of DARM. These experiments continued the approach to identifying deficiencies in DARM, but served mainly as an opportunity to determine dependability characteristics of DARM, which again is used to compare DARM to similar frameworks.

1.1 Organization

Chapter 2 briefly introduce Spread and DARM, as well as some of the already established vendors of cloud computing, and how DARM relate to these. Chapter 3 describes the work that had to be done to achieve a better development platform on DARM. Experiments are explained in Chapter 4. Chapter 5 presents results and discussion. Conclusion and future work are found in Chapter 6.

Chapter 2

Background

This chapter briefly describe DARM and its dependencies. First Spread is presented, which form a fundamental base for DARM. DARM is presented afterwards. In the end, DARM will be compared to existing cloud computing middleware.

2.1 SPREAD

Spread [1] is a high performance messaging service. It is open source (Spread Open Source License) and it provide access to define your own fully customizable group communication system. Notice that the customizing options gives opportunity to focus on one or more QoSs (Quality of Service). Included is options for guaranteed delivery, ordering, low latency, and it is also possible to combine options in order to aim for more general qualities.

Spread allow a node to join a group of nodes, given a unique group name. When in the group, messages can be broadcast (*multicast*) to every member of the group. These, among other functionalities of Spread have been exploited in DARM to enable reliable messaging and persistent states across multiple nodes.

For the challenges met here, Spread has been tuned to work in a closed environment with fast and reliable links. Spread installs by default with a very slow configuration. This is in order for it to avoid instability due to longer latencies on WAN links. Although some testing has been done with the default configuration, a much faster configuration [2] has been used mostly. This is believed to better represent a real cloud system, where the servers themselves, usually are connected by fast wired links. Wired links leave plenty of room to handle increased overhead and shorter time outs introduced by the *fast spread*

configuration. Only clients are assumed to use the slower links, which is possible since they will not intervene with the messaging between the servers.

Version 4.1 of Spread has been used for work done here.

2.2 DARM

DARM [3] build on top of Spread, utilizing the *QoS* points and customization options provided. This allows DARM to make logical decisions on the current state of any group(s) it may belong to. As long as there are available capacity in other nodes, DARM will replace any failed replica automatically onto other nodes. This is done continuously in order to maintain any minimum replica count specified. A *replica* denotes an instance of a service application deployed on a node. Each node is capable of requesting more replica for a service, the responsibility is distributed, however the node needs a leader status to do so. In a group of nodes, usually only one node will be in charge. When a leader experience a fault, or nodes are separated from their leader, a new leader will be chosen.

On a node, DARM is represented by a factory daemon. At start up it will connect to its local Spread daemon, or shut down if Spread is not already running[1]. When a DARM factory successfully connects to its local Spread daemon it will join the factory group and install the current *view* specified by the group. The *view* represents the current state of the group, listing all of its members. All factory related messaging is done within this group, including *membership messages* which is used to do concurrent and synchronous updates on the *view*. *Membership messages* may deal with joining of members, failing of members, and network partitioning.

At this point the newly instantiated node is called a *standby node* since no service has been started on it. If needed, other factories may send *create replica* requests to the newly instantiated factory. When a factory receive a *create replica* request it will start an instance of the service specified as long as it is not already running the very same service already. We say that the service is *replicated* onto the given node. Note that a service may replicate over multiple nodes, but never twice on the same node. A replica may also be initiated manually on a node running Spread and DARM.

A service utilizing DARM needs to specify certain parameters to the local DARM factory it connects to. These include *minimum replica*, *maximum replica*, *command line*, *group* and *reaction time*. A service is not allowed to connect without first specifying these

[1] A proposal to avoiding this by enabling the factory to initiate the Spread daemon was presented as future work by Joakim L. Gilje [4].

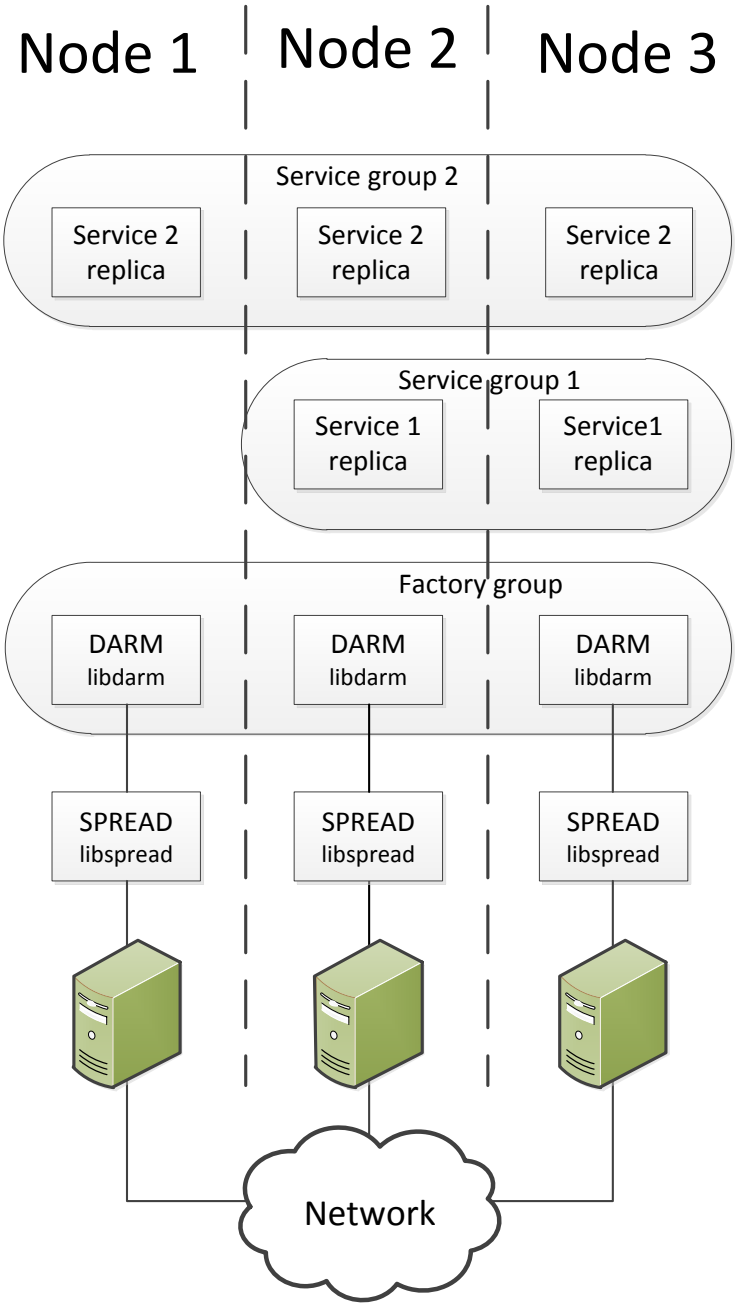

FIGURE 2.1: A network of three nodes connected though a network. Spread and DARM form a foundation for two different services.

parameters. The minimum replica is used by the factory to set the minimum number of replica for the given service. If the replica count drops below the minimum specified, the factory initiates requests for new replica to the other factories. If the number of active replica should reach the number of available factories, the factory will idle before reattempting to create more replica. The maximum replica operates in the same way as minimum replica, only the opposite way. If for example two fully deployed network partitions merge to one network, this may lead to a replica count exceeding the maximum for the given service. In this case the factory will issue a request to selfterminate associated service replica until the replica count meets the maximum. The command line parameter is simply the command line which a factory should run in order to locally initiate a new replica of the service. Also the group name to use for the service group must be given at start up. It is important that the group name is unique for the service. The factory use this group name to filter membershipmessages, which again is used to determine the number of active replica for the service. A consequence of repeated group names on different services is irregular behaviour and potentially a total failure of the service. Reaction time is a time delay that DARM will add to its actions. DARM will wait the given time (in seconds) issuing and verifying any request. This avoids incorrect assumptions that again could lead to erroneous behaviour.

DARM is able to tolerate network partitioning and its performance for such events has been documented in [5]. Service availability and other requirements will be ensured in all network partitions as DARM tries to maintain service requirements on each partition.

Figure 2.1 illustrates how Spread and DARM make the foundation for service applications. Three nodes which are all running a daemon of Spread and a DARM factory. The factories are all included in the factory group. On top of these, 2 different services are run. One service is incorporating two replica, and the other, three replica. Notice that separate groups are formed for factories and for each service. This ensures that multicast messages are only sent to members that serve the same purpose. Notice that each service replica is connected to its local Spread daemon through the DARM factory. This is not illustrated in order to keep the illustration clear.

Figure 2.2 illustrate, stepwise, how a recovery procedure in DARM proceeds. An initially active group (Group A), with three members, experience a fault on node 2. After the fault has been detected (event 2), a new view is installed at the remaining nodes; 1 and 3. The leader, situated on node 1, issues a create request (event 4) on an available node (Node 4) which soon after initializes a new replica for the given service. The new replica sends a join message to group B, resulting in a new view change with node 4 incorporated.

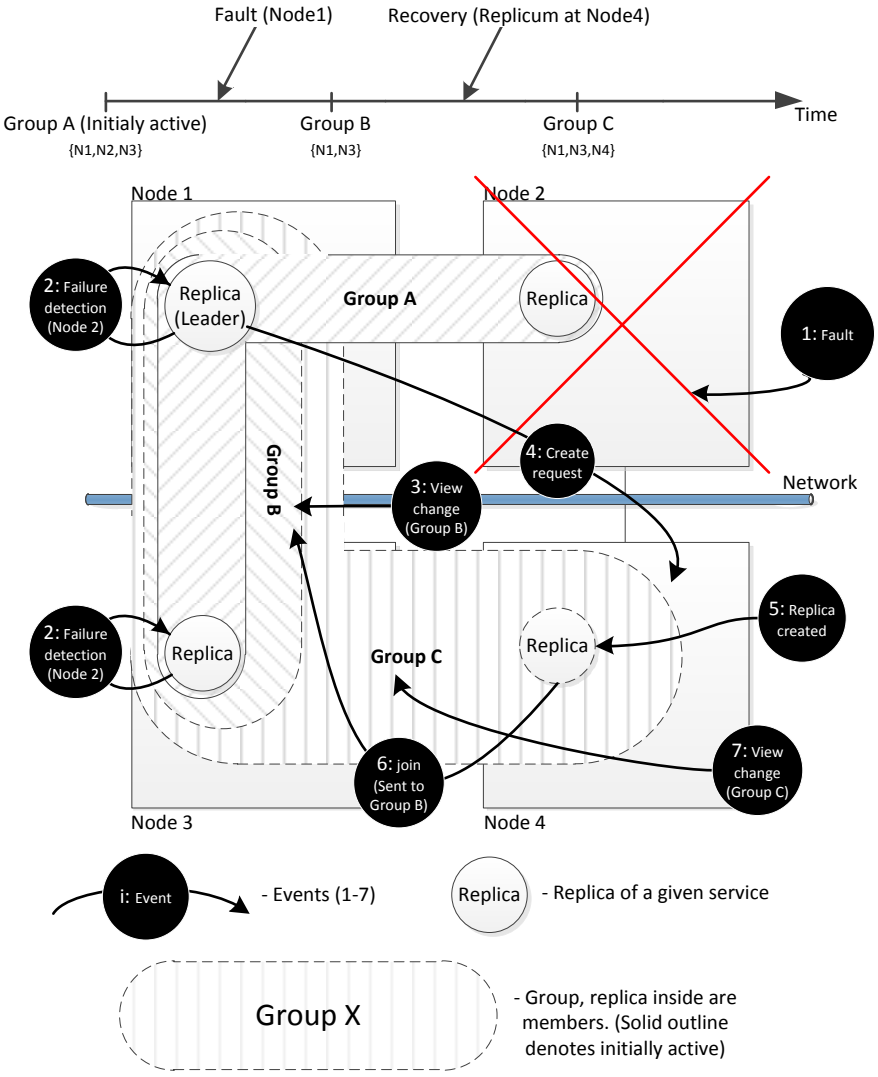

FIGURE 2.2: Illustration of a simple recovery by DARM. The time line is not representative in any way other than illustrating the order of events.

Since both Spread and DARM are written in C, a Java Native Interface (JNI) wrapper was already implemented in order to handle interaction with above layers like the Java Group Communication System (JGCS)[2] and applications. The JNI enables Java programs to interact with the C code of the underlying GCS. For a more thorough reading on how DARM exploits Spread, refer to [4].

[2]Introduced in Chapter 3

2.3 DARM and cloud computing

DARM is perhaps not yet fully developed to work as an enterprise cloud computing system. However, together with Spread, it is starting to take shape as a potential foundation, or middleware, in such a system. DARM builds on ideas [6] originally developed by associate professor Hein Meling, where the focus has been to develop a fully autonomous decentralized system in order to achieve as high as possible level of especially availability and reliability, but also scalability.

2.4 Cloud computing

Today cloud computing is settling as a foundation for large shares of Internet-based interactions. Cloud computing is becoming the norm for hosting Internet-based services. The term itself deals with delivering hosted services over Internet. As a service grow, eventually the maintaining of QoS requirements become more demanding, making it impossible for any single processor to meet the demands. Single point of failure alone is a motivator for spreading the responsibility among multiple processors. We call this process to *replicate* the service. The service experience an increase in QoS requirements, in fact most requirements are increased with such a system. This comes with a cost; increased complexity for developing.

Cloud computing denotes three main categories of service providing; *Software-as-a-Service* (SaaS), *Infrastructure-as-a-Service* (IaaS), and *Platform-as-a-Service* (PaaS). Software-as-a-Service is currently in the wind. Big cloud computing vendors are putting much effort into this field. Good examples of this are *Google*'s online office tools [7] and Google *Wave* [8]. Currently also *Microsoft* is making office tools freely available through its cloud computing infrastructure. Infrastructure-as-a-Service is generally targeting smaller companies and private persons, and usually involves renting virtual servers hosted on clouds. The last option, Platform-as-a-Service, concerns direct renting of a cloud, or usually parts of it, in order to run customized distributed applications. Each of the categories share a common strategy; they distribute among multiple nodes in order to achieve higher grades of availability, reliability, and general performance.

2.5 Actors in cloud computing

Google and *Amazon* are the most established actors in public cloud computing. With *Microsoft* currently stepping into this market as well, it becomes clear that public

cloud computing is a battleground for the giants of the IT world. Also private cloud computing is experiencing an upwind. Big and medium sized companies are employing private clouds in order to better cope with challenges in scalability and availability. Yet cloud computing is rapidly changing the field of computer science, still with room for improvement and innovation.

The number of different APIs for developing cloud computing is growing fast. Some are more developed than others. An overview over different enterprise APIs are listed here:

- *VMware* has got an API called *vCloud*. It is specially suited towards private clouds, yet it is configurable to work with, and share loads with, other public clouds. *vCloud* is a closed project, offering a 60-day trial.
- *Citrix* offer cloud developing through APIs like *NetScaler*, *Lab Manager* and more to come. Projects are closed, yet free trials are available (90-days).
- *Oracle*[2] is offering open source cloud APIs through *Sun Cloud*.
- *Google*[2] is also offering open source cloud developing through *App engine*. It allows users to define cloud based applications as private or public, yet they are all assumed to run on *Google*'s own servers.
- *Microsoft*[2] recently released a product named *Azure*. This enables users to develop applications that take advantage of clouds provided by *Microsoft*.
- *Amazon*[2] does not offer any platform for developing, however *Eucalyptus* is an open source API that resembles those of *Amazon*. This can be used to produce applications and systems very similar to those of *Amazon*.
- *Zoho*[2] offers tools to develop cloud based applications. Pre made applications allows for simple set up of *SaaS* based clouds. Everything is open source, but *Zoho* does not offer any tools to develop your own infrastructure.
- *Hadoop* is an open source API. It is used in the more commercial known project *Cloudera*. It allows to create centralized systems.
- *Enomaly* is another open source API. It focuses on delivering a convenient platform for developing private cloud infrastructure.

Most of these products have not been on the market for more than a couple of years. These actors are mentioned here to give a short overview of enterprise standards that all

[2]Some actors are only mentioned in this list because they are promoting cloud computing. They don't offer any tools or middle-ware to develop your own cloud infrastructure, only tools to create applications that may be hosted by the cloud computing vendor. This hosting usually comes with a price tag.

depend on underlying implementations of distribution algorithms. The techniques used in the different cloud systems mentioned are surrounded with much secrecy, making it hard to classify them. It is known that Amazon employ some autoscaling techniques for their clouds, yet it is not based on GCS. It might be bold to compare DARM with fully established standards, still DARM employs techniques yet to be standardized. The techniques referred to concern decentralized architecture and GCS as a bottom line. Where the vendors mentioned are focusing mainly on performance, DARM is novel in its focus on availability.

Most of the APIs mentioned in the list suffer a common weakness. They have some kind of centralized architecture. Being centralized means that they have a fixed upper limit of scalability and reliability. This is mostly due to overloading the central node. Optimizations might help avoiding this, but eventually, a continuous system growth will cause serious problems. Developing cloud computing systems has traditionally focused on centralized systems very much due to commonly accepted standards and lower grade of complexity. This implies that a higher level of reliability was only available through significantly increased cost; cost rather spent on employing manual monitoring and intervention of the service.

Cloud computing middleware are perhaps not receiving as much attention as the enterprise vendors. Also, the middleware is an area of much secrecy, and obtaining correct information can be difficult. Some of the known cloud computing middleware that are using the GCS approach to achieve fault tolerant distributed services, are presented very briefly. Many of these are based on CORBA [9], a distributed object-based platform. It allows to perform method calls on multiple nodes, yet each call on a group is actually based on multiple single calls to each member. This has potential to degrade performance. The first middleware presented is Delta-4 [10]. It provides fault treatment, but it is bound to specific hardware and operating systems, which has hindered its adoptions. FT CORBA [11] use similar techniques as those in DARM, with a generic factory, a replication manager, and a fault monitoring architecture, but it does not provide any measures to counter network partitioning. Eternal [12] is an implementation of FT CORBA that support replicating services over a network. Yet, its solutions are secret and not well known. DOORS [13] partially implement FT CORBA, but its replication management is centralized. MEAD [14] is another framework that implements FT CORBA. It offeres some simple recovery mechanisms, but it is not able to relocate replica at failures, which hinders its potential. AQuA [15] is based on parts of the CORBA standard and offers functional fault recovery. Its fault recovery procedures are costly in means of network performance and could potentially cripple the system if frequent faults are experienced. Jgroup/ARM [16] is a predecessor of DARM. JGroup extends the RMI paradigm and acts as GCS for Autonomous Replication Management

(ARM). Along with fault tolerance, it is unique in its tolerance of network partitioning. Much thanks to the underlying Spread, DARM performs better than its predecessor. The point in this brief presentation of known middleware is not to present their details, but to appoint how DARM stands out with the ability to maintain availability despite network partitioning.

Chapter 3

Applications

In order to test DARM in realistic settings, it would need applications that behave like real applications. Note that an application is the same as what has been referred to as a service prior to this chapter. A total of three applications, with clients, was needed to run the desired experiments. The applications did not exists in DARM prior to this work. From before, DARM had little support for application development. Applications would need to be built as pure spread applications, later interpreted to also include DARM support if needed. The call was for an API that could be used to develop applications, handling all interaction with Spread/DARM. This chapter will describe how a platform was developed first, and how the application has been developed on top of this platform.

3.1 JGCS

Java Group Communication System (JGCS) [17] [18] is a Java based generic interface for group communication. It allows applications to be built using a common API, and it provide bindings for a multitude of group communication systems. Figure 3.2 (Approach 2) illustrates how JGCS is placed as a layer between Spread/DARM and the applications. As Java-based group communication kits become more used, the number of applications and services based on these kits increase. The various applications usually bear many similarities, but they differ in the way they bind and use the underlying group communication system. This is where JGCS comes into play. Group communication systems bind to JGCS, and in that way promote portability to group communication applications. Application developers may use this to change the underlying group communication toolkit, while doing few or no changes to the application itself. Since different communication tool kits usually provide different qualities, this can be very useful when for instance a service requires high levels of availability or performance.

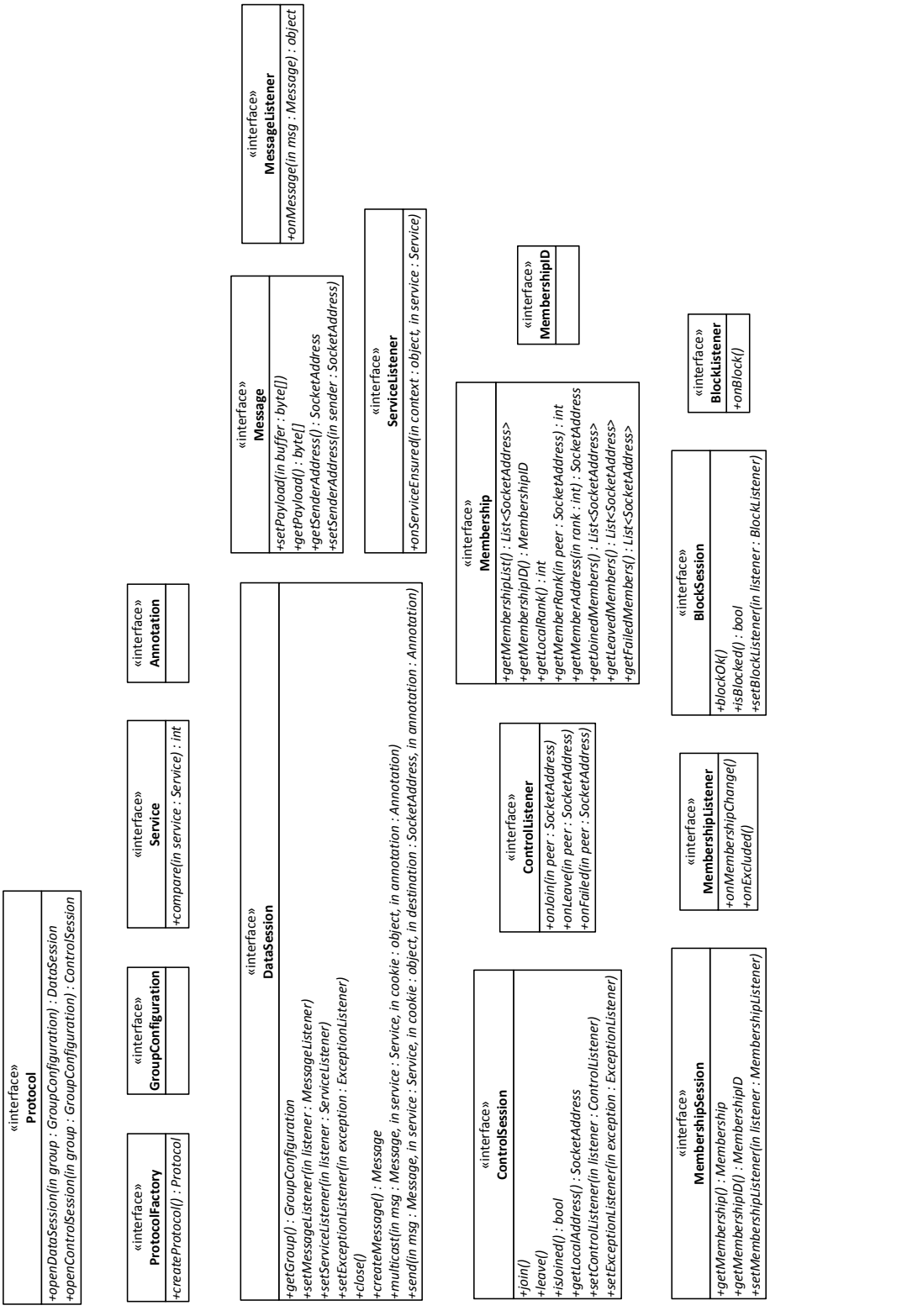

FIGURE 3.1: JGCS interface overview.

Figure 3.1 is an overview for the interfaces in JGCS. They are classified into different groups according to their purposes (see the text placed left in the diagram). Interfaces are divided into groups for easier understanding. JGCS defines its own terminology for group communications. The terminology along with how the most important interfaces are implemented, is explained in the following. The specific interfaces that will be presented are *Protocol*, *ProtocolFactory*, *DataSession*, *ControlSession*, *MessageListener*, *ControlListener*, and *MembershipListener*. The ProtocolFactory serves as the entry point for the interfaces. Service development through JGCS should start with this. The ProtocolFactory holds the parameters for the connection to the underlying GCS. The Protocol serves as an instance of a connection session. It is obtained from the ProtocolFactory. Any use of the group communication starts with the Protocol. A DataSession is used to send messages. It supports multicast invocations as well as messaging to single members. In order to receive messages, a message listener must be registered through the DataSession. The MessageListener defines a stub, triggered in the case of an incoming message which is input as an inputparameter. This stub may again be used to trigger reactions on the implementing service. A DataSession may be created through the Protocol. ControlSession is in many ways similar to a DataSession. It provides the methods needed for entering and leaving groups. In order to receive membership related messages, a listener must be registered through the ControlSession. Like the MessageListener, a ControlListener defines a stub triggered in the event of members leaving and entering the group. This stub should trigger events on the service to compensate for membershipchanges. The ControlListener needs to be registered through the ControlSession in order for proper functioning.

For DARM, the ProtocolFactory is implemented in a class called *DarmProtocolFactory*, which holds all the parameters earlier specified; minimum replica, maximum replica, groupname, commandline, and reaction time. A class called SpProtocol implements the Protocol and it is connected to the JNI wrapper of DARM.

As seen in Figure 3.1, JGCS resembles a large collection of interfaces. These could be further explained here, however it is referred to [17] for a more thorough presentation, which is a good source for further understanding of JGCS. The left hand side of Figure 3.2 illustrates how JGCS is placed between the DARM factory and the service application. Notice that also a client, on the right hand side, are required to use JGCS.

Much work has been put into implementing JGCS, and it is now fully adopted in DARM. Listings 3.1 displays the code of a simple application using the JGCS-DARM approach for connecting. The class constructor sets up the connection and joins. All the other methods are stubs from the implementation of MessageListener and ControlListener, which does not include any further functionality here for practical reasons. The code

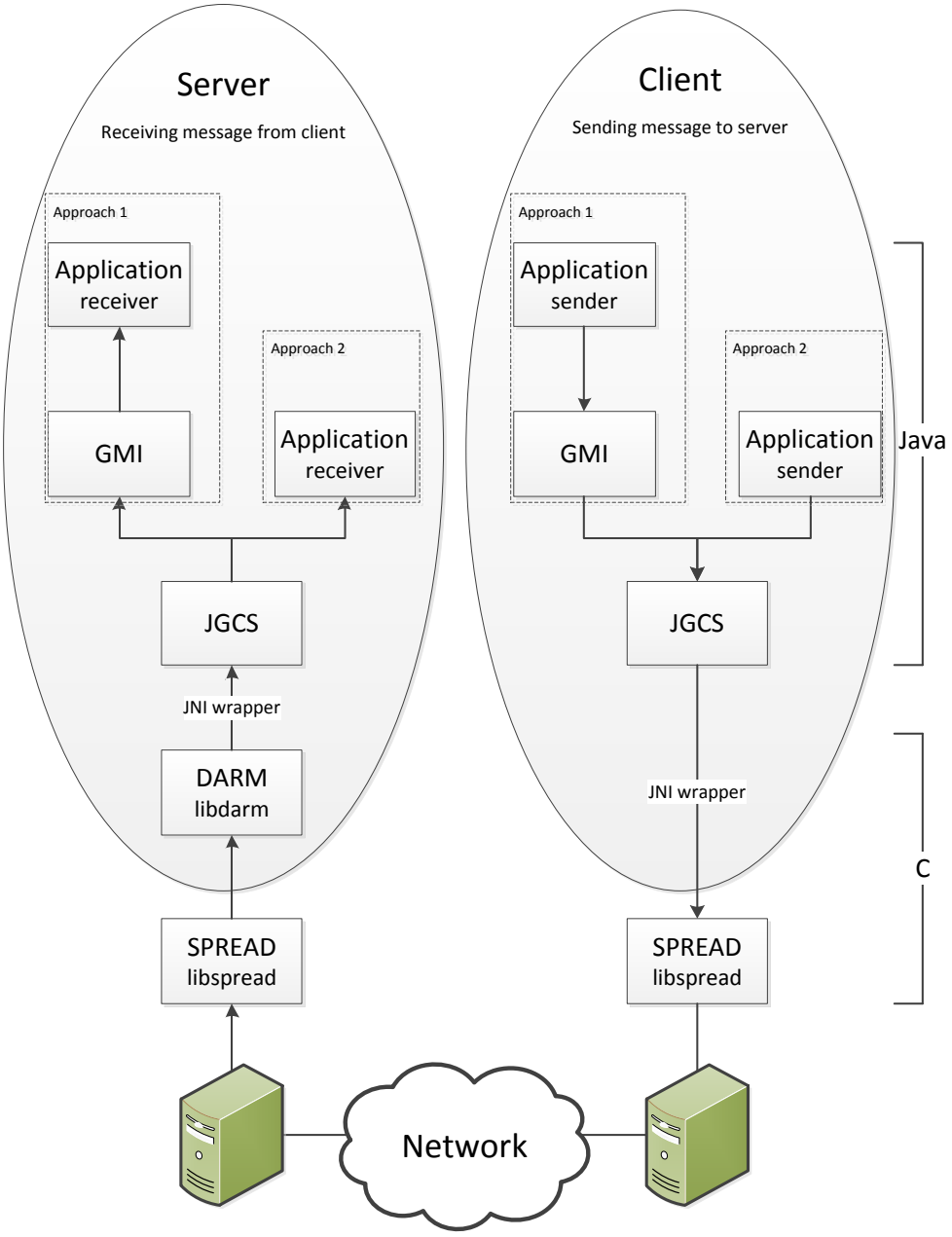

FIGURE 3.2: Typical layers for an application on a DARM network. The application use alternatively approach 1 or 2 to connect. Approach 1 use GMI while approach 2 does not.

is kept as simple as possible for easier understanding. In Listings 3.2 the same code is displayed, only with a pure Spread connection. Notice how similar they are. A pure Spread connection would for example be used in a client (without implementing the ControlListener and calling the join method on line 17).

```
1  public class Service implements MessageListener, ControlListener {
2
3    // Variables
4    private ControlSession control;
5    private DataSession data;
6
7    // Service constructor, create and join the group.
8    public Service() throws JGCSException, IOException {
9
10     DarmProtocolFactory pf = new DarmProtocolFactory();
11     pf.setcommandline("home/service/simpleService.sh")
12     pf.setMinReplicas(3);
13     pf.setMaxReplicas(5);
14     pf.setReactTime(2);
15
16     Protocol p = pf.createProtocol();
17     // GroupConfiguration is a simple holder of the group name.
18     GroupConfiguration g = new SpGroup("ServiceGroup");
19     this.control = p.openControlSession(g);
20     this.data = p.openDataSession(g);
21
22     data.setMessageListener(this);
23     control.setControlListener(this);
24
25     control.join();
26   }
27
28   // Stub for MessageListener.
29   public Object onMessage (Message msg) {
30       // Trigger message related actions.
31   }
32
33   // Stub for ControlListener, member join.
34   public void onJoin (SocketAddress peer) {
35       // Trigger member join related actions.
36   }
37
38   // Stub for ControlListener, member leave.
39   public void onLeave (SocketAddress peer) {
```

```
40          // Trigger member leave related actions.
41      }
42
43      // Stub for ControlListener, member failed.
44      public void onFailed (SocketAddress peer) {
45          // Trigger member failed related actions.
46      }
47  }
```

LISTING 3.1: A simple service application for JGCS DARM-connection (Java).

```
1   public class Service implements MessageListener, ControlListener {
2
3       // Service constructor, create and join the group.
4       public Service() throws JGCSException, IOException {
5
6           ProtocolFactory pf = new ProtocolFactory();
7
8           Protocol p = pf.createProtocol();
9           // GroupConfiguration is a simple holder of the group name.
10          GroupConfiguration g = new SpGroup("ServiceGroup");
11          this.control = p.openControlSession(g);
12          this.data = p.openDataSession(g);
13
14          data.setMessageListener(this);
15          control.setControlListener(this);
16
17          control.join();
18      }
19
20      // Stub for MessageListener.
21      public Object onMessage (Message msg) {
22          // Trigger message related actions.
23      }
24
25      // Stub for ControlListener, member join.
26      public void onJoin (SocketAddress peer) {
27          // Trigger member join related actions.
28      }
29
30      // Stub for ControlListener, member leave.
31      public void onLeave (SocketAddress peer) {
32          // Trigger member leave related actions.
33      }
```

```
34
35    // Stub for ControlListener, member failed.
36    public void onFailed (SocketAddress peer) {
37        // Trigger member failed related actions.
38    }
39 }
```

LISTING 3.2: Code for *JGCS SPREAD*-connection (Java).

Version 0.6.1 of JGCS have been used.

3.2 GMI

Group Method Invocation (GMI) is a tool available to use in combination with group communication systems. It build on ideas described in [16]. Similar to Remote Method Invocation (RMI), GMI allows a developer to seemingly call methods directly on a remote object. In GMI the remote object may be represented by a group of nodes, not just a single remote object like in RMI. The GMI API implemented here involves a server side proxy which, in few steps, allow a client to do *anycast* invocations. Anycast invocations is when a message is transmitted to a random member of the group to which the client connects. This is useful in order to achieve load balancing.

GMI has been implemented in DARM with abilities to completely handle the connection to both Spread and DARM. An example of how to set up a connection through GMI is presented in Listings 3.3. The *ServerSideProxy* class of GMI handles the rest. Notice how a DARM-connection requires extra input parameters (line 8), minimum replica, maximum replica, and reaction time, similar to those seen in Listing 3.1 (line 11-14). After the *join()*-method (line 15) of *ServerSideProxy* has been called, the proxy object (line 4) may be used to do group method invocations on the server group members.

```
1  public Service (String name, String command, int minReplicas, int maxReplicas, int reactTime, boolean darm)
2      throws Exception {
3
4    private ServerSideProxy proxy;
5
6    if (darm) {
7      // Let GMI handle DARM connection
8      proxy = new ServerSideProxy(this, name, command, minReplicas, maxReplicas, reactTime);
9    }
10   else {
```

```
11      // DARM less connection, or handled by application
12      proxy = new ServerSideProxy(this, name, port);
13    }
14
15    proxy.join("ServiceGroup");
16  }
```

Listing 3.3: Code for GMI-connection (Java).

In order for a client to not join the server group through GMI, the *join()*-method would not be called. The proxy object (*GroupProxy*) can still be used to send and receive messages, yet the client will not receive membership messages, and the servers will not interpret the client as a member of the view.

3.3 The applications

A short description of the applications with their functionalities are presented here. Note that each application is built in a distributed fashion, allowing unlimited number of application replica to connect and together provide the services offered. This allows DARM to start and stop any replica without paying much attention to the application. However, each application specify a minimum and maximum number of replica sought, when connecting to the local factory. Note that these values may not yet be changed post deployment.

The clients should be able to produce uniformly distributed loads. Preferably they should also be able to run as threads in order to scale up the loads generated with a limited number of nodes in the environment. Clients are generally not configured to join any group. It will not receive any membership messages, or messages meant for the server group. Even though it is out of the scope of this work, security is increased.

3.3.1 Mail Service

The mail service is a partial port of an application initially designed as a pure Spread application. Due to a high dependency to the way it connected to Spread, the mail service basically required a total reconstruction. As the name says the application resembles a simplified mail service. Clients are allowed to register and send messages that are stored and available for any receiver at a later stage. The mail service does not take full advantage of GMI. GMI is only implemented to allows clients to connect to a random mail server through anycast. Also the connection to DARM factory is handled

by GMI. Most of the other functionalities are preserved, specified in the application source itself (pure JGCS-Spread functionality).

At start up, a mail server will check if other servers are online and request an update on the current client data. If no server is online the data is loaded from locally persistent data. Throughout the lifetime of the mail service, servers will keep themselves updated on any data changes by interacting with each other.

A complete list of functionalities available to clients are listed below.

- **login** is used to authenticate the client. Since the simplified mail service does not apply any security, login is really used as a register function. If the user does not exist, it will be created.
- **getInbox** gets all the message headers with corresponding information like date, sender, receiver as well as an attribute saying if the mail has previously been read.
- **send** is used to send messages to other registered users. The message is stored and remains available until the receiver requests its deletion.
- **read** is used to read the whole message. The message is marked as read when this request is received at the server.
- **delete** is used to delete a specified message.
- **changeServer** is used to reacquire an address to a random available server. It is automatically called whenever the client is started, or when a server becomes unavailable. It is the only method run through GMI.

3.3.1.1 Mail Client

A client has been designed to produce loads in a random manner. The client logs on, reads messages, and then send mails to other users stored in a list. In order to prevent too big amounts of mails accumulating on the service, a client delete some of its messages when the inbox reach a certain size.

Any cast is only used in order to connect to a random server. Once connected the client will not change server as long as the server is available of change

3.3.2 Allocator Service

The allocator service resembles a resource allocator, for example a DHCP server. Like the mail service, this program is based on existing solutions and has been ported into

this project. The application was already known through the course *MID110 Distributed systems* at UiS, where it functions as an introduction project to Spread. The allocator service was already using the GMI package, which handles all message passing in this application.

Servers share a pool of resources equally. Responsibility changes when servers join or leave. Clients are allowed to request some of the resources. If the resource is available, it is leased to the client for a given amount of time. Clients may refresh the lease, and servers may reacquire resources when the lease time has expired.

In this example a resource is simply defined as an address. However, what the service actually governs is not considered of any importance, only that its functionality resembles those of a real allocator system. Changes could easily be implemented to suit other demands.

3.3.2.1 Allocator Client

Clients of an allocator service only deal with two types of requests. All of the client messaging is done by any cast.

- **getAddress** is used to acquire an instance of the resource. If the server has no available resources it will try to reacquire any expired leases, or request from the other servers. If no resources are available, this is reported to the client.
- **refresh** is used to refresh the lease time of a resource. It is usually called when the existing lease time drops below some defined point. If the refresh request is any cast to a server whom is not the owner of the lease, the server will forward the request the owner before replying if a refresh has occurred or not.

The client has been designed to run as a thread, which allows to run multiple clients per node. This has been done to produce even bigger loads on the servers.

3.4 Transaction Service

The transaction service was not successfully incorporated during this work. It has previously been used in DARM. Yet some difficulties with reusing it, due to database implementations, hindered its reuse. To fill the void, another application was used. It is only a simple application, which does not offer any services except proper logging of how it is treated by DARM. Also, no client was included for this service.

Chapter 4

Experiments

This chapter aims to explain the experiments done on DARM, and how it has been used to fully evaluate the system, with components.

Some previous work has been done on this field. DARM was measured in Joakim L. Gilje's master thesis [4] [5]. These measures concern recovery times when injecting single network partitions. The experiments does not include multiple services running.

Measuring was done to a broader extent on the JGroup/ARM framework [19], which is similar to DARM. The results presented in this paper are intended to be directly comparable with those presented [19], and will bear many similarities. This implies that *stratified sampling* [20] with fault injections will be used to do an overall evaluation of DARM. Advantages with this approach is that not only the platform itself will be evaluated but also all the other aspects like the services replicated on it. The approach from [19] allows to calculate probabilities and expectations for the system, which again may be used to find a mean time between *total failures* (MTBF). A total failure means that no server replica that can provide the service remains, denoted as state *U0* from state diagram in Figure 4.2. The state U0 is typically achieved when the service experience node failure on all nodes currently possessed, before being able to recover. A total failure currently requires manual intervention for recovery.

All experiments have been performed under somewhat controlled environments at facilities provided by UiS. A set up of 20 nodes have been used in different ways to produce and measure various realistic scenarios. The nodes are identical, each running an Intel core2 1.8 GHz CPU with 2 GByte of RAM. Only Linux has been used in this project (CentOS 5), but both Spread and DARM are compatible to run on all of the most popular operating systems. The nodes actually also serves as workstations at UiS. This has introduced potential risks of uncontrolled workloads generated at the nodes.

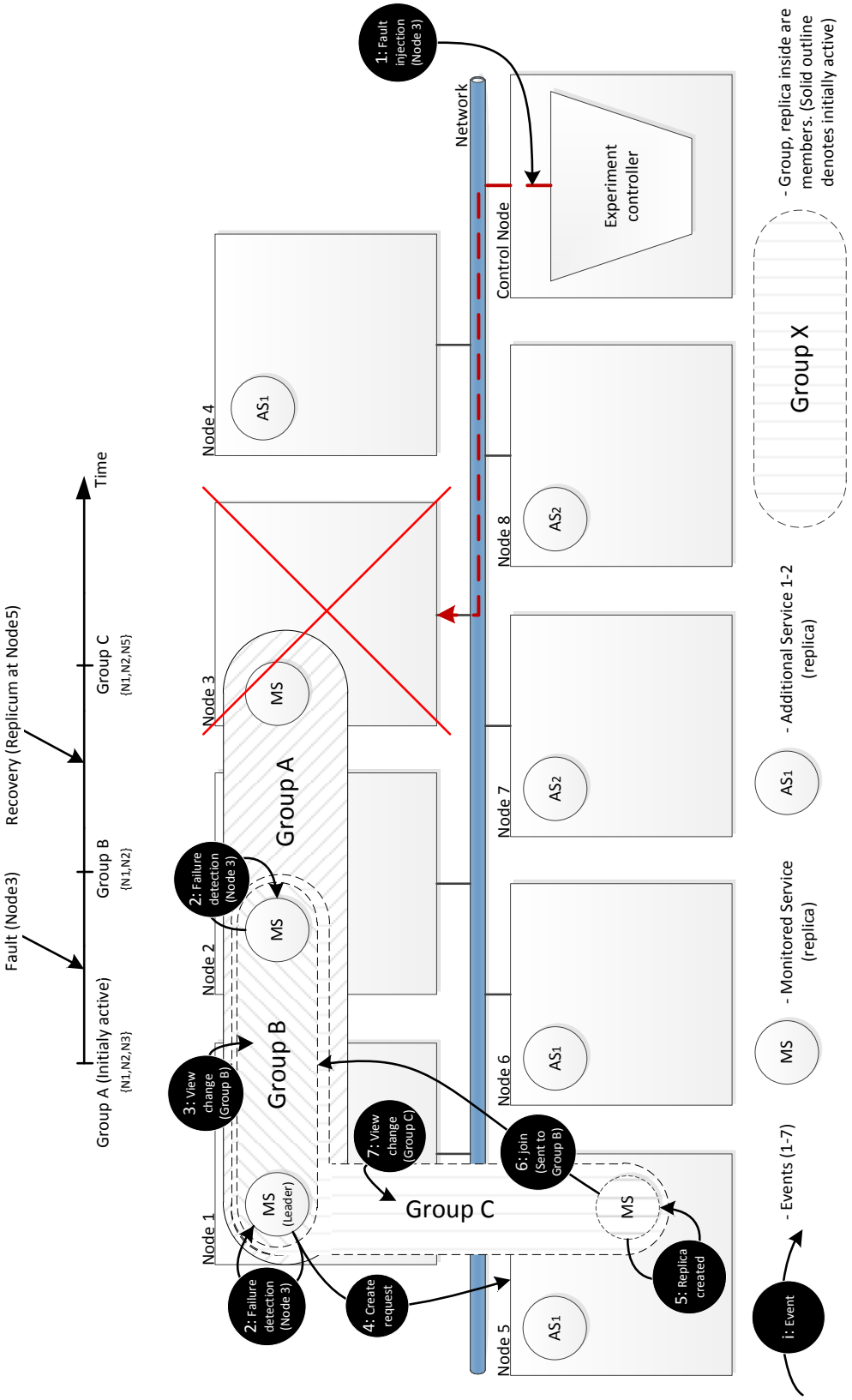

FIGURE 4.1: Experiment set up, with failure injection illustrated. The time line is not representative in any way other than illustrating the order of events.

4.1 Stratified Sampling

Shortly summarized stratified sampling concerns looking at a subsystem in stead of looking at the system as a whole. The system is then exposed to one of multiple strata. A stratum in this case concerns a run with n faults. A stratum with three consecutive faults is denoted $Strata_3$. Notice that the strata classification is actually based on the number of faults experienced before being able to recover. Looking at a subsystem, this enables us to compute dependability characteristics of the system as a whole. The process will be further explained in the following sections. The monitored subsystem is referred to as the *MS*. MS resemble the subsystem. Figure 4.1 illustrates how the MS, along with two AS_n (Additional Service), are replicated over multiple nodes. In the figure a fault (event 1) is injected, and the following events are all consequences of the first event, occurring as necessary steps toward recovery.

More details on how stratified sampling has been used with DARM are presented in Section 5.4 later in this chapter.

4.2 State Machine

In order to produce the data needed to compute the dependability characteristics, it is important to define a set of states that resemble the possible states of the subsystem. This is directly implemented in the log parser, and it does help when meeting some of the complexity there. Figure 4.2 illustrates the state machine. It is very similar to the state machine presented for JGroup/ARM in [19]. This one is somewhat simplified because some of the behaviour in [19] is not observed for DARM.

As mentioned the state machine is used to identify the states of a monitored service experiencing various fault injection schemes. In order to complete calculations, later presented, time spent in each state is analysed from the log data. It is especially the time spent in any of the U (service **U**navailable) states that is used later for calculations. With the defined states it is possible to present a set of events as a set of states. We call this a *trajectory*. As long as there have been no resent faults or network partitions for a given service, it will be situated in the *stable* state A0, which resembles the base state. In fact, A0 is the state of which a service will normally spend most of its time. States A5 and A9 are also considered as stable states, the state machine assumes the service is configured with minimum and maximum replica set to three, and it will therefore not settle in any other state than A0 as long as three or more nodes are available.

An example of a fault trajectory will be presented. It is assumed that the given service is initially situated in A0. When a fault occurs, the service loose one of its three replica and drop to state A4. Soon after a viewchange occurs, and the state change to A5. Since minimum replica is set to three, it requests a new service replica. When the new replica is starting up, but not yet involved in the view, the service is in state A1. Soon a viewchange is initiated and the service is recovered back to A0. This trajectory may be described as a trajectory set, $X_i = \{A0, A4, A5, A1, A0\}$.

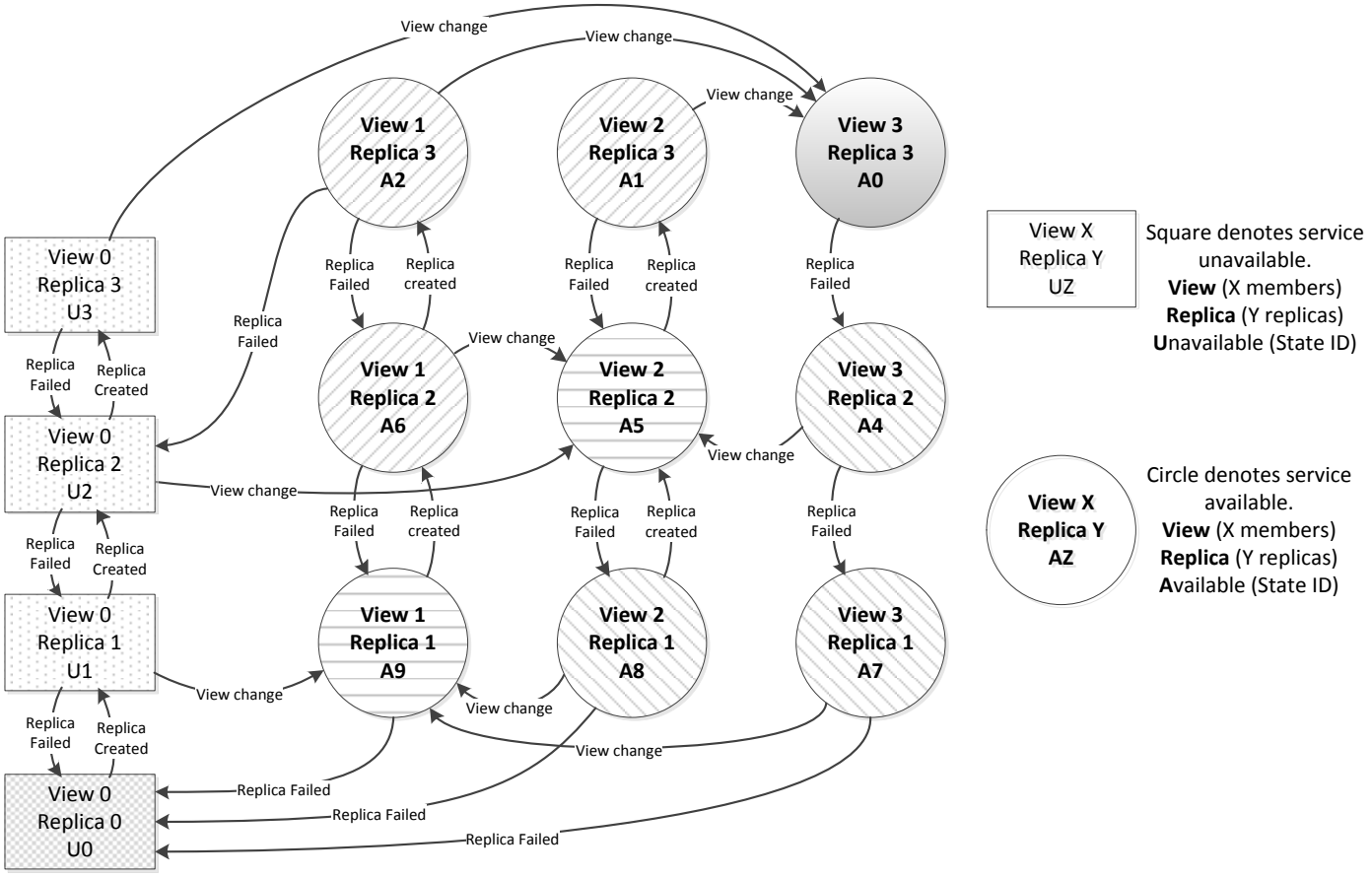

FIGURE 4.2: State machine for a service requesting 3 replicas.

4.3 Experiment Controller

The experiments have mostly been controlled and run by bash scripts. This is a simple solution, yet it offers the accuracy needed for experimenting. Bash scripts simply store command lines in a sequence. When a bash script is executed from a terminal later, all command lines are executed according to the sequence they are stored. Any command

may be run indirectly from a bash script in stead of directly in the terminal. The standard logic operators are also supported in bash scripts. This means that bash scripts may actually be used from a control node to *SSH* (Secure Shell) into other nodes, and in that way control an experiment spanning over multiple nodes.

Some experiments involve shutting down one or more random nodes, this is also referred to as *injecting a fault*. A random node is selected according to a list of all nodes in the experiment. The control node starts a shutdown script on the target node which kills all Spread and DARM related processes.

Bash scripts have performed as expected. Two issues with this approach are the accuracy of sleeping, and deviation between logged event and actual time of event. When multiple faults are injected in one experiment iteration, the bash script should sleep for a random time between the injections. When sleeping involve millisecond accuracy, deviations of up to almost 0,05 seconds have been observed. Average deviation settles around 5-7 milliseconds. The sleep deviation means that the first fault will never occur before the experiment has run for at least 3 milliseconds. Also the time between two fault injections are believed to never drop below 3 milliseconds. When logging the time for an actual fault injection, the script performs a little slow. Therefore it is believed that a deviation at about 5 milliseconds exists between time of fault injection and actual time of injection.

The deviations presented are not handled in any way, mostly because the divergence presented is small, and they are believed to neutralize each other in the long run.

When logs from multiple nodes are compared, another issue arise with how synchronous the system clocks at each node are. *NTPStat* (Network Time Protocol Statistics) has been used to monitor any big deviations. It is used to print an overview of each node's offset according to an *NTP* server. Every 30 iteration of long lasting experiments includes a print with offsets from each node. The offset rarely exceed a millisecond, except for some nodes that seem to have more fluctuating system clocks than others. These have been excluded from roles which could lead to any implications on the experiments. No routine for offset compensation has been implemented, yet it is easy to spot any big deviations through the logs. Time synchronization have occurred regularly through experimenting.

4.4 Dependability

The theory behind dependability analysis of autonomous replicated systems is presented in [19] where stratified sampling together with fault injections are used to estimate dependability attributes of JGroup/ARM. The same strategy has been adopted in

this experiment. It has been attempted to use the same parameters, and set up the experiment in the same way, in order to produce directly comparable results.

A total of 9 nodes have been utilized, where one node is not a participant as it runs the experiment controller. Three distinct applications are used, two of them are set up to replicate over three nodes, and one application claim two replicas. A total of 8 replica are divided among 8 nodes.

This experiment is divided into three *sub experiments*. First 1000 iterations are run with one fault injection per iteration, then 1000 with two fault injections, and finally 1000 iterations with three fault injections. Except from the number of faults, each iteration is identical for each of the sub experiments. Each service is configured with a reaction time of 2 seconds. Post experiments, it was found out that this diverged from the experiments in [19] which used 3 seconds for reaction time.

An iteration is built up as follows:

1. **Initialization** is the first phase. It first starts a Spread daemon and a DARM factory on each of the 8 involved nodes. They are now what is referred to as *standby nodes*, not running any applications, yet accepting any create replica requests. Next, the applications are started. The applications are supposed to automatically evenly distribute among each of the available nodes[1]. A time delay is injected here to ensure the system is stable before moving on to step 2.

2. **Fault injection** is the phase we are interested in. The current status of each application is logged. One, two, or three unique nodes are selected randomly, before a fault is injected on each. Faults are injected randomly at times drawn from the interval $[0 - T_{max}\rangle$, where T_{max} has been set to 15 seconds. In order to make the node chosen for fault injection inoperative as soon as possible after the fault injection is initiated, the Spread daemon is target first. As long as there are no Spread daemon running on the node, the communication link to the other nodes are broken. Next the factory and the service replica on the node is shut down. After all faults have been injected, a delay of T_{max} seconds is imposed to allow for stabilizing, before moving on to step 3.

3. **Shut down** is the final phase of an iteration. The shut down involves shutting down the Spread daemon, the DARM daemon, and every replica on each node in the experiment setup. At some stages of experimenting this phase was resulting in erroneous iterations due to replica not being shut down properly, continuing

[1] Due to a bug in DARM, a tendency that applications stacked up on some nodes, with others unused, left no other choice than manually distributing the applications at start up as it would leave unwanted effects on the statistical properties. See future work for a more thorough explanation of this issue.

into the next iterations. This has been countered by running a shut down script twice, once after the iteration end, and once more 60 seconds after the iteration end. This has worked all the time, yet it introduce a great increase in the time required to run the full experiment (60 seconds * 3000 iterations).

4.4.1 Logging

Each node produce log files while running; factory- and replica instances. The replica logs are collected, post experiment, and woven together according to the time of each log entry. Since each node is more or less synchronized, we are able to store each event in one common file, while maintaining a chronological order. The events are organized by time of occurrence, in milliseconds. This is done for each sub experiment, and it produces three larger log files which eventually holds the total activity log. After another algorithm has tested the actual stratum of each iteration, the total log files are parted into smaller files, iterations, and put into folders representing each set of strata. Recall that what happens between the first fault injection until the first full recovery, state $A0$, decides the actual stratum. If for example a *2-fault-injection* iteration is stabilized before the second fault is injected, this iteration is actually classified as $Strata_1$.

```
885927|iteration:268 has started.
888254|pitter7|spreadWasShutDown
888604|pitter10_MailService|failed|#67452466#pitter9|viewcount=2|FailsBy:pitter7
888604|pitter9_MailService|failed|#67452466#pitter9|viewcount=2|FailsBy:pitter7
890476|pitter13|spreadWasShutDown
890853|pitter15_AllocatorService|failed|#89515834#pitter11|viewcount=2|FailsBy:pitter13
890855|pitter11_AllocatorService|failed|#89515834#pitter11|viewcount=2|FailsBy:pitter13
890955|pitter5_MailService|startingup
891004|pitter5_MailService|join
891015|pitter10_MailService|onJoin|#2880755#pitter5|viewcount=3
891016|pitter9_MailService|onJoin|#2880755#pitter5|viewcount=3
891021|pitter5_MailService|onJoin|#2880755#pitter5|viewcount=3
891737|pitter13_AllocatorService|terminated
891936|pitter9|spreadWasShutDown
892266|pitter10_MailService|failed|#2880755#pitter5|viewcount=2|FailsBy:pitter9
892267|pitter5_MailService|failed|#2880755#pitter5|viewcount=2|FailsBy:pitter9
892957|pitter10_AllocatorService|startingup
893010|pitter10_AllocatorService|join
893023|pitter15_AllocatorService|onJoin|#21969985#pitter10|viewcount=3
893024|pitter11_AllocatorService|onJoin|#21969985#pitter10|viewcount=3
893024|pitter10_AllocatorService|onJoin|#21969985#pitter10|viewcount=3
894410|pitter15_MailService|startingup
```

```
23 894448|pitter15_MailService|join
24 894466|pitter10_MailService|onJoin|#95293024#pitter15|viewcount=3
25 894467|pitter5_MailService|onJoin|#95293024#pitter15|viewcount=3
26 894471|pitter15_MailService|onJoin|#95293024#pitter15|viewcount=3
27 906949|iteration:268 has ended
```

LISTING 4.1: Sample log file for a strata3 trajectory.

Listings 4.1 is a sample log file for a $Strata_3$ trajectory. Notice how each single line or log event is started with the timestamp (in milliseconds). A *pitter* with a following number denotes the name of the node. The first fault is injected on node 7 approximately 2.3 seconds after the iteration is started (line 2, "spreadWasShutDown"). Line 3 and 4 denotes the following viewchange for the stricken service (MailService). Notice that it is the leader who initiate the viewchange, and the leader is given in the log event (pitter9) along with an event identifier. A new fault is injected after approximately 4.4 seconds on pitter13 (line 5). On line 8 we see that a new replica for the MailService is initiated on pitter5. It is incorporated in a new MailService view that installs on lines 10-13. When studying this log file, we also see that a total of three faults are injected before each service is able to recover to three replica per service. The recovery time for this trajectory is calculated by subtracting the last recovery event on line 26, denoted t_f, with the time for the first fault injection, denoted t_s. Hence the recovery time for this trajectory is 6.217 seconds, denoted T.

Now it is finally possible to use the files to produce the data needed to calculate dependability attributes like down times, system failure, and unavailability.

4.4.2 Formula

The goal of calculations done on the data produced by the experiment is to compute the probabilities, unavailability metric and the system MTBF. This section will first present and briefly explain the steps involved to determine the measurements for the experiments approach. Secondly a fixed approach to determine measurements without determining expectations through experimenting will be presented. The later is done for a comparison, and it uses fixed expectations calculated through somewhat realistic assumptions. This way of achieving a comparison is inspired by [19] which use the same approach.

Different formula, with short descriptions are presented here. For a more thorough explanation and proofs, refer to [19]. In order to calculate the dependability characteristics, through stratified sampling, it is important to trace the MS. Time spent in each state

is recorded and make up the data needed for dependability calculations. If a trajectory visits one of the unavailable states, the time duration of service unavailability is recorded. However states, U1, U2, and U3 have not been observed for the experiments except for the initialization phase. U0 is observed. When an U0 occurs the service will have no chance to recover without manual intervention, and a fixed downtime of 5 minutes is added to the downtime for each occurrence.

The expectation, $E(T)$, for the system is calculated per strata. When analysing the experiment, the sum of all recovery times divided on the number of trajectories for the given strata makes up the expectation. Along with expectation, the total variance σ is calculated and used to find the standard deviation $sd = \sqrt{\sigma}$. Standard deviation is calculated per strata.

For a comparison, the measurements have also been calculated through certain assumptions on the same setup as described for the experiments. This have been done in two somewhat different ways. One is to assume processor recovery, where a recovery is done by an automatic reboot of 5 minutes. The other is assuming a manual processor recovery, which is estimated to 2 hours. The challenge in calculating measurements for a fixed system, is to find the probabilities for the different strata. Expectation is given by Θ_i where i denotes the trajectory. Equation 4.1 is for calculating expectancy where p_i is the unknown probability for trajectory i. We can classify the trajectories into 3 groups, one for each strata. A strata k is denoted S_k, where k is the number of failure events in the trajectory. The probability of a trajectory in S_k is represented by the sum of all probabilities for trajectory i, denoted π_k which is displayed in Equation 4.2. Since we are only considering a maximum of three concurrent node failure events, we are able to derive equations for the needed π_k.

$$\Theta = E(T) = \sum_{\forall i} p_i T_i \tag{4.1}$$

$$\pi_k = \sum_{\forall i \in S_k} p_i \tag{4.2}$$

An equation for π_2 is displayed in Equation 4.3 where λ denotes the failure intensity, and n is the number of nodes involved in the system. Notice that λ is an estimate for all calculations it is involved in. However two different set of calculations are provided. We may say that $\lambda = 1/pmtbf$, where pmtbf is mean time between processor failure (in milliseconds). It should not be mistaken with MTBF which denoted mean time between total failure of the system. The two different set of calculation use respectively pmtbf set to 100 days, and pmtbf set to 200 days. Notice that Equation 4.3 introduce π_1 as a

factor. Here we use an estimate and assume that since the probability of a trajectory falling into a higher strata than $strata_1$ is realistically very low, π_1 must equivalently be very close to 1, hence we say that $1 \approx \pi_1 \approx 1 - \pi_2 - \pi_3$. In other words π_1 is set to 1. When π_2 and π_3 have been calculated, we revert and get a more accurate estimation for π_1 through Equation 4.5.

$$\pi_2 = (n-1)\lambda\Theta_1\pi_1 \tag{4.3}$$

$$\pi_3 = (n-1)(n-2)\lambda^2(\Theta_1\Theta_2 - 1/2(\Theta_1^2 + \sigma_1))\pi_1 \tag{4.4}$$

$$\pi_1 = 1 - \pi_2 - \pi_3 \tag{4.5}$$

The service unavailability is denoted as $\hat{U}$ and can be obtained through Equation 4.6 where $E(Y^d)$ is the sum of expected downtime for the given experiment set. In the experiment approach, $E(Y^d)$ is achieved through adding up the actual downtime for the MS, but for the fixed measurement it is obtained through multiplying π_1 with the expected downtime; time for processor recovery, or time for manual recovery. MTBF is obtained through Equation 4.7 where $E(Y^f)$ is the expected probability of failure for the given experiment set. It is determined by dividing the number of occurences of unavailability (U states) with the total count of $Strata_3$. MTBF is only calculated for the experiment approach.

$$\hat{U} \approx E(Y^d)n\lambda \tag{4.6}$$

$$\hat{\Lambda} = 1/MTBF \approx E(Y^f)n\lambda \tag{4.7}$$

4.5 Latency

As previously explained, clients have been designed to generate loads on the services. Each client logs its requests with latency, the time it takes before a reply. While a steady load was exercised, a fault was injected on one of the service replicas. This is interesting because the logs will reveal if the fault injection will have any implications on the latencies logged. Also this can be used to determine unavailability not observed in the service logs. For example it may be assumed that when latency increase to 1

second the service may be defined as unavailable. This approach could potentially be used to reveal if a service should request a higher level of replication in order to increase its availability.

This sub experiment has been fully adopted in DARM. Yet work remains to present the results, and it is therefore added to future work. The sub experiment should be added to the dependability experiment in order to completely confirm the availability grade of DARM through fault injections. Latencies observed through experiments done, points towards little affection while fault and recovery occurs. They are averagely ranged from 8-17 milliseconds on a fast network, with spikes reaching up to 100 milliseconds close after a fault injection. However uncertainty surrounds these results as they have not been confirm or checked in any thorough way.

Chapter 5

Results

The results presented in this chapter mainly focus on the stratified sampling experiments done with DARM. The chapter is aimed to presenting the results achieved as well as comparing them with the result obtained for JGroup/ARM in [19].

The strata distribution is presented in Table 5.1. Although each sub experiment has been run with 1000 iterations we see that the distribution has rapidly changed. After 1000 *3-fault-injection* iterations have been analysed, 584 fall into $Strata_1$, 289 fall into $Strata_2$, and 110 classify as $Strata_3$. Again, the reason for this is that the system is able to stabilize into state A0 before the next fault is injected. A more evenly distribution could be achieved by setting T_{max} lower; resulting in more condensed fault injections. This has not been done in order to maintain the comparability with [19]. Also this point towards a faster performance of DARM compared to JGroup/ARM since more often the system is able to stabilize before next fault injection. The fact that experiments in [19] used 3 second reaction time, while these experiments use 2 seconds potentially enables the DARM to recover more rapidly, and puts an uncertainty to the previous statement. This again leads to a higher degree of 2- and 3-fault-injections falling into lower strata classifications. This has been problematic as a lower number of $Strata_2$ and $Strata_3$ forms a somewhat poorer foundation for expectation calculations. The

Fault injections	Iterations	Strata	Distribution
1	1000	$Strata_1$	998
2	1000	$Strata_1$	683
		$Strata_2$	302
3	1000	$Strata_1$	584
		$Strata_2$	289
		$Strata_3$	110

TABLE 5.1: Strata distribution after analysis.

Classification	Count	$\theta_k = E(T\|S_k)$	sd $= \sqrt{\sigma_k}$	θ_k, 95% conf.int.	Highest	Lowest
$Strata_1$	2265	2569.22	478.23	(1631.89, 3506.55)	16659	1742
$Strata_2$	591	4158.83	1039.10	(2122.18, 6195.47)	12869	2496
$Strata_3$	110	5966.58	1550.90	(2926.82, 9006.35)	16086	3046

TABLE 5.2: Statistics from recovery times (in milliseconds)

experiment should be rerun[1] with adjusted parameters; lower T_{max} and reactiontime set to 3 seconds. This would however differentiate it from [19] and they would not be comparable.

Notice that the iteration count does not equal the corresponding strata distribution. This is, for the most part, due to iterations that does not successfully recover. Also for $Strata_1$, 2 iterations are erroneous. One (the only for all 3000 iterations) fail to properly initialize due to a bug[2]. This bug will not cause the service to fall into any of the unavailable states, and probably the bug would resolve itself over time. Yet the iteration is classified as a failure for all services. The second 1-fault-error is due to improper initialization, where only a subset of the available nodes are used for the initial deployment. The later iteration is rejected as an iteration with no fault occurrences. For $Strata_2$ there are 15 occurrences of the U0 state. All of these are caused by two fault injections occurring in short intervals, $[0.1 - 2.1]$ seconds, on the service with two replica only. The short fault injection interval leave no room for recovery an this service reach U0. For $Strata_3$ there are one occurrence of the same bug as presented for $Strata_1$, which is classified as leaving all services in U0 even though it actually leaves the services available, yet not correctly. $Strata_3$ has two occurrences of U0 for the MS. Faults are injected through intervals of total 1 and 0.9 seconds on all its three replica. Also $Strata_3$ has 14 occurrences of U0 for the service with two replica with intervals ranging $[0.5 - 2.2]$ seconds. Note that the second service running three replica (not MS) never experience the U0 state in these experiments.

5.1 Probability Density

First, the log files have been used to generate probability density graphs. For each iteration the time from first fault injection till the time of first *full recovery* is collected and stored in a set. This is done for each stratum collection. A full recovery is when all included services reach their specified minimum replication (State A0). Recovery

[1] A rerun of the experiment is listed in future work.

[2] Multiple service replica on one node is a bug presented in future work. It defies the rule that a service should not replicate twice on the same node

times are collected separately for each set of strata. Table 5.2 hold statistics from the experiments.

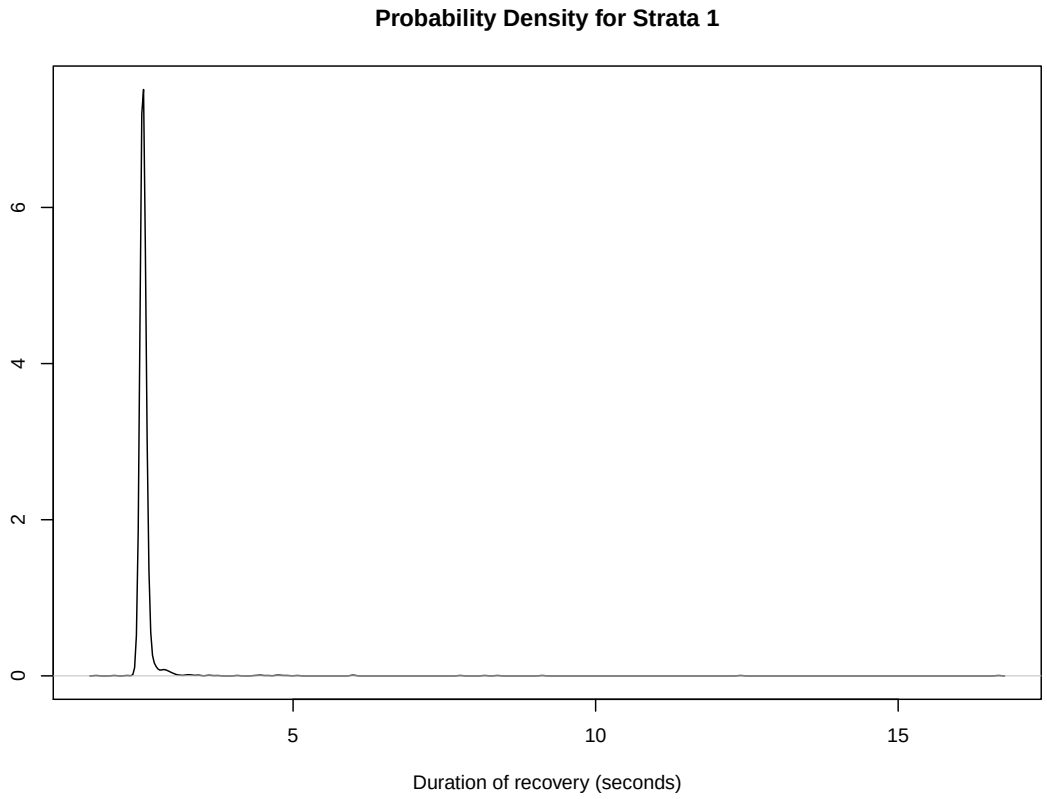

FIGURE 5.1: Duration for failure trajectory in $Strata_1$.

Figure 5.1 presents the probability density graph for $Strata_1$. The graph has a very clear peak, but with some outliers. Two outliers stand out at 12.394 (occurence O_1) and 16.659 (occurence O_2). These have been confirmed with the log manually. When studying the log files for both occurrences it is observed that the increased recovery stems from residence in state A1. The logs confirm that a replica is starting up, but have not yet initiated a join request. This process usually takes less than 100 milliseconds. O_1 spends 6.1 seconds in this state, and O_2 close to 8 seconds. During the O_2 postponing of joining, DARM even initiates a new replica request, which results in a service replica overflow (4 replica) shortly after. The overflow is compensated by a replica termination afterwards. This analyse indicate that extended recovery periods are caused by reasons out of reach for DARM. Most probable CPU/IO starvation is the reason for the increased recovery periods observed, which again support the value of service replication. Another reason for increased recovery times could be delays introduced by the membership service in Spread. The mean of $Strata_1$ is 2.569 seconds. The density graph for $Strata_1$ is very sharp, hence the expectancy for a $Strata_1$ recovery is very probable to fall close to the mean.

Compared to JGroup/ARM, DARM generally achieves a full recovery almost 5 seconds faster than its predecessor.

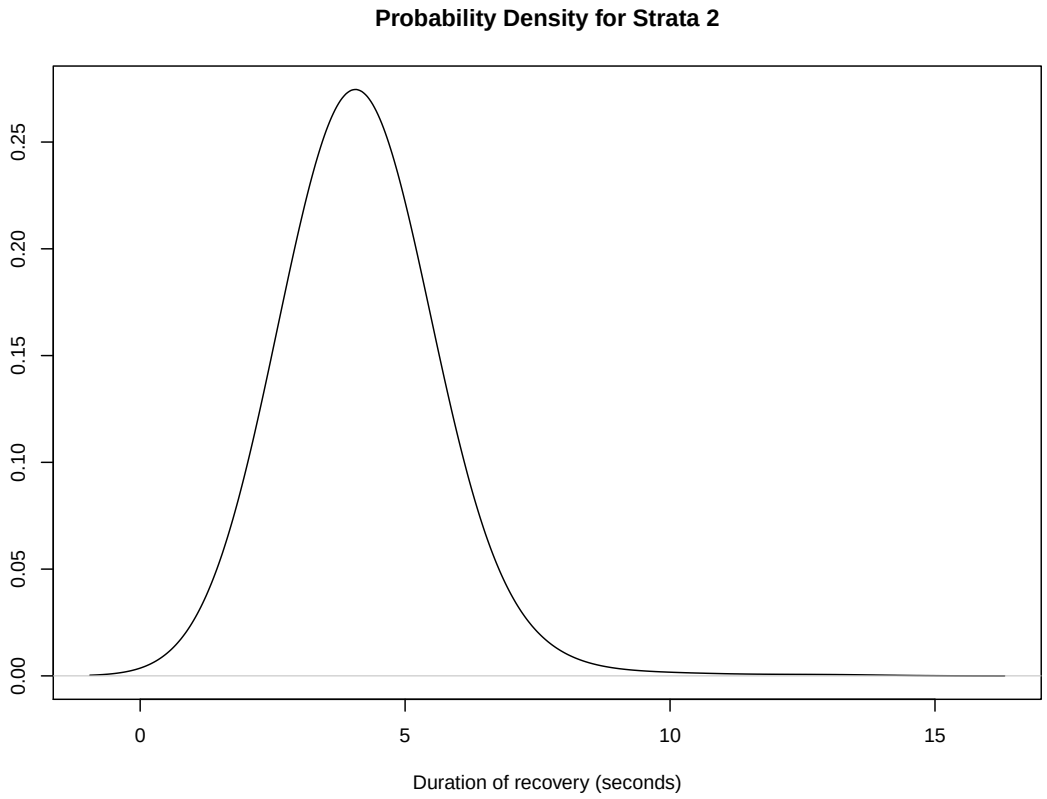

FIGURE 5.2: Duration for failure trajectory in $Strata_2$.

Figure 5.2 presents the probability density graph for $Strata_2$. We see that the expectancy is more spread out, compared to Figure 5.1 for $Strata_1$. This is expected as the different trajectories for a $Strata_2$ is more variable considering that the two fault injections occur at different time intervals. The highest values of $Strata_2$ recoveries are 10.439 and 12.869 seconds. They have both been manually checked against the logs in suspicion of the same cause for increased recoverytimes as observed for $Strata_1$. However the explanations for these values seems to be that the interval of failures have been close to the maximum of what DARM "allows" without recovery taking place in between. The mean value of a $Strata_2$ recovery is 4.158 seconds.

The highest value observed for a $Strata_2$ recovery in DARM is only 0.1 seconds above the mean value of $Strata_2$ recovery in JGroup/ARM. Again the performance of DARM is proven better than that of the JGroup/ARM framework, the variance however is somewhat the same for both frameworks.

Figure 5.3 presents the probability density graph for $Strata_3$. The graph is somewhat misleading since its lower bound covers areas below its lowest observed value. Notice

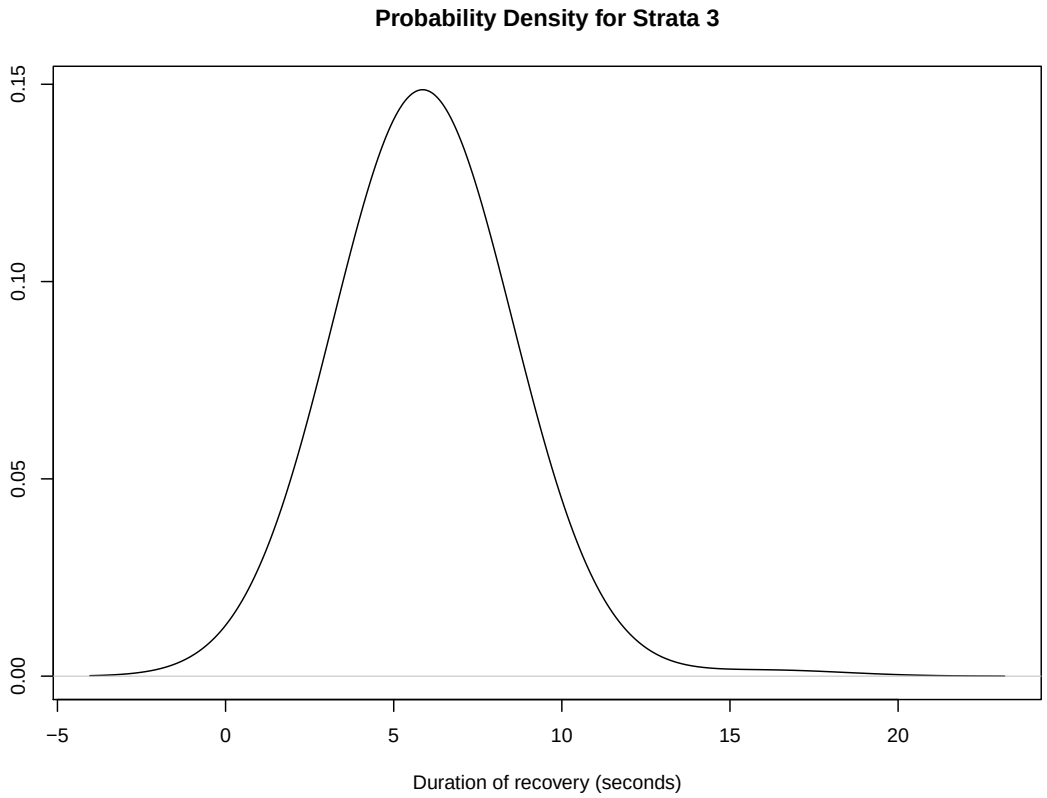

FIGURE 5.3: Duration for failure trajectory in $Strata_3$.

that no observations is done below 3.046 seconds. The mean value of a $Strata_3$ recovery is 5.966 seconds. One outlier stands out from the others, with a value close to double of the previous; 16.086 seconds. The main source of the increased recovery time seems to be a very long stay in state A5. Soon after the replicafailure a viewchange occurs. But from here it takes 10.659 seconds before a new replica is starting up. It is hard to tell the reason for this from the logs, as replica requests are not logged. However a leader change has occurred in the given service group sometime between the viewchange, from A4 to A5, and the viewchange from A1 to A0. My guess is that this along with possible CPU/IO starvations have together contributed to the extra recovery time observed.

5.2 Dependability characteristics

The results from the experiments have been used to calculate unconditional probabilities for the different trajectories to classify as the specified strata, unavailability metric ($\hat{U}$), and the MTBF ($\hat{\Lambda}^{-1}$). Notice that the MTBF is only liable for the MS. For example, the less replicated service would not score the same value as it experience a far higher count of U0 occurrences. The sections processor recovery and manual processor recovery

	Experiment Recovery Period		Processor Recovery (5 min.)		Manual Processor Recovery (2 hrs.)	
	Processor Mean Time Between Failure (pmtbf=λ^{-1}) (in days)					
	100	200	100	200	100	200
π_1	0.9999979184	0.9999989592	0.9997568889	0.9998784583	0.9941238281	0.9970726237
π_2	$2.0815438 \cdot 10^{-6}$	$1.0407719 \cdot 10^{-6}$	$2.4305555 \cdot 10^{-4}$	$1.2152777 \cdot 10^{-4}$	$5.8333333 \cdot 10^{-3}$	$2.9166666 \cdot 10^{-3}$
π_3	$4.0903937 \cdot 10^{-12}$	$1.0225984 \cdot 10^{-12}$	$5.5447048 \cdot 10^{-8}$	$1.3861762 \cdot 10^{-8}$	$4.2838541 \cdot 10^{-5}$	$1.0709635 \cdot 10^{-5}$
$\hat{U}$	$4.1317108 \cdot 10^{-17}$	$5.1646385 \cdot 10^{-18}$	$2.7771024 \cdot 10^{-4}$	$1.3887200 \cdot 10^{-4}$	$6.6274921 \cdot 10^{-3}$	$6.6471508 \cdot 10^{-3}$
$\hat{\Lambda}^{-1}$	212 yrs	851 yrs	-	-	-	-

TABLE 5.3: Computed probabilities, unavailability metric and the system MTBF.

of Table 5.3 lists the calculations done for the fixed but comparable system. The two bad runs, one from $Strata_1$ and one from $Strata_3$, along with two occurrences of U0 for the MS, make up the foundation for $E(Y^d)$ and $E(Y^f)$ presented in Equation 4.6 and 4.7 respectively.

The MTBF of 212 and 851 years indicate that DARM has become more stable than its predecessor. It also indicate that DARM should be considered a highly reliable platform for service replication, with much potential.

Chapter 6

Conclusion

This thesis has presented the work that has been done to expand the usability of DARM. Service development done on the expanded DARM platform has already proven useful. Service development on DARM has become more available to newly introduced developers with an increased user-friendliness. Tools that solve challenges that are considered highly important for distributed services are now incorporated in DARM.

This work has confirmed that any remaining deficiencies in DARM is only revealed through extensive testing. As in previous work, employing stratified sampling on a fault tolerant replication system has proven to be a an effective approach for such testing. The approach is made available in a flexible and almost fully automatic code. Its reuse may prove valuable for later versions of DARM, or when DARM is deployed in different environments.

The results achieved through this work has showed that DARM performs faster than the similar JGroup/ARM system. DARM performs almost flawlessly, with only rare events of potentially service disrupting behaviour. A mean time between failures of 212, indicate that DARM performance is stable and that it provides very high capabilities for availability and reliability. It has been indicated that the main contributor to service failures is actually the rare events of multiple consecutive node faults. However, even though the occurrences of erroneous behaviour are very rare, they indicate that DARM still has room for improvements and concrete deficiencies in the framework have been pointed at.

6.1 Future Work

Further ideas for improvement on the DARM framework is presented here.

6.1.1 Issues in DARM

Multiple issues remain in DARM. Someone with sufficient C programming skills should indulge on these challenges. Some of the issues I have stumbled across are listed:

- *Distribution* usually does not occur with ideal distribution. This issue reveals itself especially when starting multiple services simultaneously without manual placement.
- *Multiple replications on the same node* have been seen from time to time. Especially when the number of available nodes, become less than the number of desired replications for a service has this problem presented itself.

6.1.2 Unavailability as seen from clients

As mentioned, this experiment has been readily developed in DARM. Yet it has not been run properly, so it is considered an easy extension to this thesis. However a concrete measure for how large latencies should be allowed, before the service is classified as unavailable, must be defined. This could typically be set to one second, or even less. The latencies that have been observed through simple 1-fault-injections have pointed towards little affection on the service availability experienced by clients during reduced service replication and recovery actions.

6.1.3 Experiment rerun

A rerun of the experiments should be done. The basis for this simple task is already laid. The logging tools could be used to analyse the logs, resulting in a very minimum of work needed to readily prepare the results. The rerun should be run with a lower T_{max} and a reactiontime set to 3 seconds in order to produce a more evenly distributed collection of strata occurrences. Also the dependability as seen by clients should be incorporated in this rerun.

6.1.4 Measuring on network partitioning

This point depends on one of the issues remaining in DARM. Due to network partitioning eventual resulting in partition with few nodes, the "multiple replications on same node" issue has prevented this work from being done. Experimenting with network partitioning combined with fault injection could prove useful for further seeking out of deficiencies not yet observed for DARM.

6.1.5 Tools for DARM

We have looked into GMI in this thesis. It is meant as a tool kit for service developers. Other tool kits could be developed as well.

- *Database* could be handled by one such tool. It should offer developers to easy add synchronous databases for their services, and even allow for services to rely on distributed concurrent databases.
- *Security* per service would also soon present itself as a desired addition to DARM. Encryption could be handled by a tool kit for fast implementation. This could for example be solved by a third party key provider. The key provider could be a service implemented in DARM.

6.1.6 Administration console for DARM

It would be useful to have an administration console for DARM. Some of the console functionalities present tasks that would inflict security issues on DARM, which would need to be handled first. The console should enable an administrator to do different tasks:

- An *overview* should allow to monitor the current status of the DARM system. All nodes with current replica should be listed.
- The *shut down* of a DARM service can be a tedious task as a shut down must be initiated on all replicas of the service simultaneously. Especially when one want to stop one service while inflicting no influence on already deployed services, does this process present itself as complicated. This task should be handled by the factories by request. Malicious nodes could use this to acquire adverse capabilities, so it would require security measures.

6.1.7 Live reconfiguration

This issue is also presented as future work in [4]. I wanted to readdress this issue since it could be implemented as a tool set for DARM applications, much like the GMI package presented in Chapter 3.

6.1.8 More generic experiments

The experiments seen in this thesis should be available to run on different environments. This would require a better solution to running the experiments themselves. As explained, this is currently solved with shell scripts. These are not very portable and include some solutions that are not portable to other operating systems. A proper way to achieve this could be to build a Spread based experiment manager. Since Spread is highly portable to other operating systems, and it is able to shut down or exclude single nodes, it could prove ideal for such a task. The Spread based experiment engine is also able to inject network partitions through already developed tools like SpMonitor [1]

6.1.9 Optimal mapping

The placement of replica onto nodes (mapping) is hardly in focus in DARM. Proper techniques for achieving this have been developed and described. One such technique is the Cross Entropy Ant System [21] (CEAS) which uses a distributed and decentralized approach to find optimal mappings. DARM could prove as a good fit for such a system. The optimal mappings are extracted through the use of foraging ants that are passed between the nodes.

Bibliography

[1] SPREAD. The Spread Toolkit, June 2001. [online] `http://docs.google.com`.

[2] Very Fast Spread. URL `http://commedia.cnds.jhu.edu/pipermail/spread-users/2004-July/002114.html`.

[3] DARM. A framework for Distributed Autonomous Replication Management, 2007. [online] `http://darm.ux.uis.no`.

[4] Joakim L. Gilje. Autonomous fault treatment in the spread group communication system. Master's thesis, Unversity of Stavanger, June 2007. URL `http://darm.ux.uis.no/papers/gilje-msc-thesis.pdf`.

[5] Hein Meling and Joakim L. Gilje. A Distributed Approach to Autonomous Fault Treatment in Spread. In *Proceedings of the 7th European Dependable Computing Conference*. IEEE Computer Society, May 2008. URL `http://darm.ux.uis.no/papers/meling-edcc2008.pdf`.

[6] Hein Meling. An Architecture for Self-healing Autonomous Object Groups. In *4th International Conference on Autonomic and Trusted Computing*, volume 4610 of *Lecture Notes in Computer Science*, pages 156–168, Hong Kong, China, July 2007. Springer-Verlag. doi: 10.1007/978-3-540-73547-2_18.

[7] Google Docs. Online Google office tools, September 2007. [online] `http://www.spread.org`.

[8] Google Wave. A new web application for real-time communication and collaboration, May 2009. [online] `http://wave.google.com`.

[9] Object Management Group. The common Object Request Broker: Architecture and Specification, Rev 2.3., June 1999. Object Management Group, Framingham MA.

[10] David Powell. Distributed Fault Tolerance: Lessons from Delta-4. *IEEE Micro*, pages 36-47, February 1994.

[11] Object Management Group. Fault Tolerant CORBA Specification. OMG Technical Committee Document ptc/00-04-04, Object Management Group, Framingham, MA, April 2000.

[12] Louise E. Moser, Peter Michael Melliar-Smith, and Priya Narasimhan. Consistent Object Replication in the Eternal System. *Theory and Practice of Object Systems*, 4(2):81-92, January 1998.

[13] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. DOORS: towards high-performance fault tolerant CORBA. In *Proceeding of the 2nd International Symposium, Distributed Objects Applications (DOA)*, pages 39-48, Antwerp, Belgium, September 2000.

[14] Carlos F. Reverte and Priya Narasimhan. Decentralized Resource Management and Fault-Tolerance for Distributed CORBA Applications. In *Proceedings of the 9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, 2003.

[15] Yansong Ren, David E. Bakken, Tod Courtney, Michael Cukier, David A. Karr, Paul Rubel, Chetan Sabnis, William H. Sanders, Richard E. Schantz, and Mouna Seri. AQuA: an adaptive architecture that provides dependable distributed objects. *IEEE Transactions on computers,* 52(1):31-50, January 2003.

[16] Hein Meling, Alberto Montresor, Bjarne E. Helvik, and Ozalp Babaoglu. Jgroup/ARM: a distributed object group platform with autonomous replication management. *Software: Practice and Experience*, 38(9):885–923, July 2008. doi: 10.1002/spe.853. URL `http://www3.interscience.wiley.com/cgi-bin/abstract/116325377/ABSTRACT`.

[17] JGCS. Java Group Communication System. A generic interface for Group Communication, 2006. [online] `http://jgcs.sourceforge.net/`.

[18] Nuno Carvalho, Jose Pereira, and Luis Rodrigues. Towards a Generic Communication Service. In *Proceedings of the 8th International Symposium on Distributed Objects and Applications(DOA)*, September 2006. URL `http://jgcs.sourceforge.net/_Media/doa-06-13.pdf`.

[19] Bjarne E. Helvik, Hein Meling, and Alberto Montresor. An approach to experimentally obtain service dependability characteristics of the jgroup/arm system. In *EDCC*, pages 179–198, 2005.

[20] Michel Cukier, David Powell, and Jean Arlat. Coverage estimation methods for stratified fault-injection. *IEEE Trans. Comput.*, 48(7):707–723, 1999. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/12.780878.

[21] Máté J. Csorba, Hein Meling, and Poul E. Heegaard. Laying pheromone trails for balanced and dependable component mappings. In *IWSOS '09: Proceedings of the 4th IFIP TC 6 International Workshop on Self-Organizing Systems*, pages 50–64, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-10864-8. doi: http://dx.doi.org/10.1007/978-3-642-10865-5_5.